RUST

Patrick Gilmore
Department of Computer Science
University of Arizona
Tucson, Arizona 85721
pmgilmore41@email.arizna.edu

Jack Lyons
Department of Computer Science
University of Arizona
Tucson, Arizona 85721
jacklyons@email.arizona.edu

April 6, 2020

ABSTRACT

1 Introduction

2 History

2.1 Why?

Most systems applications are written in C or C++ but as many masters of C know, C can have tons of Pitfalls. To solve this problem Rust was developed so that developers could write high Performance code that would normally need to be written in C or C++. Now with Rust developers need to worry much less about memory failure and code failing as a result of a segmentation fault.

2.2 Who?

The development of Rust was started by Graydon Hoare, a self proclaimed language engineer, in 2006 has a personal project. Down the line Mozilla took an interest in the project and put together a team to assist in the development of Rust. Mozilla continued to back Rust with the aim to rebuild their browser stack. To do so Mozilla aimed to replace the difficult C++ code with something that was easier to debug, write, and maintain efficiency.

2.3 Rust Today

Rust is still going strong to this day as indicated by stackoverflow.com, a popular site for developers to learn and ask questions about technology. In fact stackoverflow has voted Rust to by the most loved language 4 years in a row. Rust's drive to reduce the number of pains that developers face while writing system code in C++ while minimizing the consequences has been pulling in frustrated developers for years. So long as a developer does not need to make an application that goes way deep into the fundamentals of a system, Rust will permit easy convenient code.

3 Control Structures

Given that most large applications will require some degree of control flow, it would only make sense that rust would include some of its own. Since rust is derived from languages like C and C++ some of the structures may look familiar. That is not to say that rust has no unique features of its own!

3.1 If-else

Of all the control structures found across the many programming languages in existence, the if-else structure is the most common. For those less familiar, the if-else structure is used to evaluate the truth (aka Boolean value) of some

statement and determine was block of code to execute on. This can help programs only perform certain tasks when the correct conditions are met and avoid code that could crash the application if the conditions are not met (like a null). Rust takes a pretty standard approach to how these structures look.

```
let x = 5

if x == 5 {
    println!("x is 5");
}
else {
    println!("x is not 5");
}
```

The structure takes some statement, determines the truth of it and jumps to the block of code that is appropriate. In the case above the statement is true as x is equal to 5 so it will jump to the block of code starting just below the if clause. It is important to note that in rust the statement MUST be a Boolean or at least evaluate to a Boolean. Programming languages like C allow the clause to be a number but if that was attempted in rust, the program would fail.
Rust also supports the common else if statement which allows a program to step through a number of possible options while only choosing one.

```
let x = ??

if x == 5 {
    println!("x is 5");
}
else if x < 5 {
    println!("x is less than 5");
}
else {
    println!("x greater than 5");
}
```

The above code will start by checking if x is 5 and if it is not then it will check if x is less than five and finally assume it must be greater than 5.
A very interesting feature of the Rust if-else structure is its ability to assign variables on the clause. We likely see statements like the following in the more common coding languages:

```
let mut x = 5

if x == 5 {
    x = 10
}
else {
    x = 2
}
```

But in rust we have the ability to assign x from the if statement itself!

```
let x = 5

x = if x == 5{
    10
}
else {
    2
}
```

The only catch to this neat little trick is that in each block, the value assigned to x needs to be of compatible types. So, if we had tried to assign x to "two" in the else block an error would have triggered.

3.2 Loop

Rust comes with an interesting control structure called loop. Loop will repeat until the program is stopped manually or until a condition within the loop block is met. This is not like for loops you would see in other languages where the loop is executed however many times and the condition is looping on changes is constant manner at the end of each iteration (i++). Rust does include a while loop which achieves its purpose in a very similar manner so whats the point. Well Rust allows you to set a return value to the loop.

```
let mut count = 0;
let returnCode = loop {
    count = count + 1;

    if count = 5 {
        break count;
    }
}
```

The above code would perform the loop until the count variable reaches 5 and then returns that number into the new variable returnCode. This may prove useful to those who need a variable quickly figured out based on dynamic data but then never change afterwards.

3.3 while

This may come as a very refreshing structure to those learning rust as it is pretty much the same as you would find in any other language. The while loop combines the power of loop, if, else and break and proves to be a useful tool in most programs. At the beginning of the structure, the programmer sets a condition for the loop to end on and follows that with the body of code they want repeated. It can allow loops to be shortened much more if the return of the statements in the code block are unneeded.

```
let mut count = 0;
while count != 5 {
    count = count + 1;
    println!("Count is {}", count);
}
```

In the above code we don't care about what count is, we just want the program to print what it is at each step so that mean the return code is useless to us. A perfect time to us the while loop!! It is important to remember to have a section within the block that changes the state of the loop so that at some point the program to exit and not get caught in an infinite loop.

3.4 for

Programmers will often find themselves needing to iterate over a data set and determine some meaningful result from the data within. This is where the for structure comes in! Much like loop and while, for performs a block of code on repeat until a condition is met forcing it to exit. Unlike the other two, for must be over an iterable type like a list. It is still possible to perform loops similar to what we saw in the while section using for. It simply requires the use of ranges in Rust. Since the range is iterable, Rust can step through it until it reaches then end.

```
let mut lst = [1,2,3,4,5];
for elem in lst.iter() {
    println!("Count is {}", elem);
}

for num in (1..5) {
    println!("Count is {}", num);
}
```

The loops above both do the same thing but the first is over a list which could contain in theory anything while the second uses a range between 1 and 5. Since for tends to achieve a large subsection of what programmers need to get done, it is the most popular of the looping structures.

3.5 match

Another of the major control structures available in Rust is match. Match is very similar to the switch case structure you would find in C or Java. The idea behind it is that you provide a case (something to match to) and a series of rules for the possible cases you care about. A use for this would be to have a list of rules based on student grades. The case would be a letter grade and the definition for the case could be the GPA equivalence. This is very convenient for what would normally be massive if else blocks as they do the same thing but with much less syntax required. As we mentioned this is very similar to the switch case structures found elsewhere but with major differences in the syntax.

```
let x = 2;
match x {
    1 => {println!("1");},
    2 => {println!("2");},
    _ => {println!("unknown");}
}
```

The above is equivalent to:

```
switch(x) {
    case 1:
        print("1");
        break;
    case 2:
        print("2");
        break;
    default:
        print("unknown");
}
```

Each case is the value before the '=>' and the definition for the case follows. It should be noted that the brackets in the definition section are not required if the definition is only one statement. To represent the default case, the case you go to if all else do not match is represented by and underscore which means anything. The Rust syntax also does not require the break to signal the end of a case rather it uses a comma.

4 Data Types

As is the goal with many programming languages Rust performs a variety of operations over a range of available data types put into the language. These data types are important to understand as Rust is a Strong and Statically typed language. This means that all types are known at compile time and it is very difficult to have the types interact in such a way that might create a failed program. In this section we will explore the data types available in Rust and how they can be used to write a meaningful program.

4.1 Integer Types

The integer is a very common type found across multiple programming languages and can also be found here in Rust. Rust allows for a greater degree of specificity of the integer types and in fact requires it! In computers integers are represented by a certain number of bits, most often being 32. Rust permits a range of these Integers sizes to allow programmers to get values they need while controlling the amount of space they are using. The sizes available are; 8, 16, 32, 64, 128 as either signed or unsigned Integers. To represent these in variables the programmer should follow this general structure:

```
let [mut] name: [i/u][size] = ...;
```

Rust also offers an isize and usize that determines 32 or 64 bit number depending on the computers architecture which is useful for if the programmer wants the ability to get the largest possible integer. To represent these integer values Rust can display them as decimals, octals, hex, binary and byte.

4.2 Floating-Point

Floating-point numbers are not all that different from the integer type except that they can only come in 32 and 64 bit configurations. To any unfamiliar with floats, they essentially are how we represent non-integer number (numbers with

values to the right of the decimal point). Users find them often in the computation of real data like grades, expenses, averages, ect. The only other significant point to make with floats is how you declare them in your Rust code:

```
let [mut] name: f[32/64] = ...;
```

4.3 Bool

Bool types may also be referred to as Boolean types as it is how it is referred to as in other contexts. Bool is simply either true or false. This can be assigned outright as declaring a variable to be equal to true or false as well as from comparative operations. To declare a bool simply:

```
let [mut] name: bool = [false/true];
```

4.4 Character

Another of the common variable types that Rust shares with most of the programming world is the character type which is most commonly just a letter. In Rust the character is a 4 byte Unicode scalar value which encompasses all letters as well as letters found in other languages not English and can even support emojis. To specify a variable to be a char you simply need single quotes like so:

```
let var = 'A';
```

4.5 Tuples

Sometimes a single variable of one type just is not enough to get the job done which is where Tuples come in. Tuple are data types that can merge together a collection of variables of whatever type the user requires. A tuples size is determined by the user though often tuples are interpreted to have 2 values and the definition of each slot in the tuple defined by the programmer as well. Consider the following:

```
let tup = (i32, f32, u8) = (604, 4.5, 8);

let (x,y,z) = tup;
```

The above is a tuple of a 32-bit integer, 32-bit float and 8-bit unsigned integer. Notice how we went ahead and declared an arbitrary tuple and set it equal to the first tuple. This allows us to access each slot of the tuple by the variable in the respective index. Another valid option would have been to call the original tuple with a dot operator and the number of the desired index.

4.6 Arrays

Arrays are a point were Rust attempted to fix some of the headache of C. Before we get to that, an Array is a fixed collection of data all of the same type. Each value in the array can be accessed by its index within the array starting from 0. Now we are gonna bring back around why Rust made arrays nicer than C. In C if you were to attempt to index into an index outside the bounds of the array, it would grab the memory it reaches from the pointer of the array plus index. Rust does not allow that to happen by killing the program with a run-time error it determines to use because it checked if the desired index was in bounds before accessing. Take a look an example below:

```
let arr1 = [1,2,3,4];
let arr2: [i32, 5] = [1,2,3,4,5];
let arr3: [3, 5];

let var = arr1[0];
```

All the array declaration above are valid but do so in special ways. The first just creates an array pre-filled and the type is inferred at compile time. The second has the programmer both declaring the type it should have within the array and the length it should be. The third has what the programmer would like to be the default value for the array and the size (ie that one would be [3,3,3,3,3] which means the compiler infers the type from the default value. Lastly examine the final line that demonstrates accessing the array. Arrays in rust, being fixed and on the stack, are random access and the program can simply jump to the desired index.

4.7 Structs

Coming over from C is the infamous struct. Structs are a custom data type created by the programmer that contain a variety of types (maybe even other structs). These often are used to represent a collection of related data such as a student record or employee data. To create one of these structs begin with the key word Struct followed by the name of the new type and curly brackets that contain a comma separated list of the types within the struct. Once declared making an instance of the struct is much simpler than it was in C where the programmer needed to malloc space for the struct and begin filling it. This can be done as mimic of the declaration but the curly brackets filled with the desired values or each variable within the struct can be accessed with a dot operator on the struct variable.

```
Struct Student {
    name: String,
    Grade: 'A'
    Graduation_Year: u8,
}
```

Here we have declared the struct for students that keeps record of their name, grade and graduation year. Then we need to make an instance of the struct:

```
let John = Student {
    name: String::from("John Deer"),
    Grade: 'B',
    Graduation_Year: 2022,
};

John.Grade = 'C';
```

Above we instantiated a Student struct but assumed something with their grade changed so we accessed it from the new struct and modified its value. Also notice how Grade in our declaration was not given a type but rather we gave it a default value and the compiler will then infer the type from that default to be Character. To anyone who want to make meaningful programs with Rust, practicing with Structs is vital as they are key to handling large collections of records.

4.8 Std Library

We have talked about a lot of data types that are available in Rust but most of them are primitive types that cannot be broken down further but Rust like Java has a number of special std Library types including String and Heap. Like String, many of these types are on a heap and have the ability to grow with demand. In C Strings had to be represented as a character array but that array was fixed in size. With Rusts library, Strings were introduced and could grow beyond the original size of the sequence of characters. A similar principle applies to the other major data types found in the std Library. What is very important about these types is that there are library functions associated with these types to perform special operations on each type. These types all prove to be very useful in many applications because their ability to change as the program executes is helpful and much less difficult than you would have found in similar structures in C or C++.

5 Subprograms

6 Summary

References

[1] Community. r/rust. `https://www.reddit.com/r/rust/`. Accessed: 2020-03-31.

[2] Carlo Milanesi. *Beginning Rust: From Novice to Professional*. Apress, 2018.

[3] Rust. Control flow - the rust programming language. Accessed: 2020-03-31.

[4] Rust. Install rust. `https://www.rust-lang.org/tools/install`. Accessed: 2020-03-31.

[5] Rust. Learn rust. `https://www.rust-lang.org/learn`. Accessed: 2020-03-31.

[6] Rust. The rust reference. `https://doc.rust-lang.org/reference/index.html`. Accessed: 2020-03-31.

[7] Hugo Tunius. 5 awesome rust projects. https://hugotunius.se/2017/10/22/5-awesome-rust-projects.html, 2017. Accessed: 2020-03-31.

[2] [5] [6] [1] [7] [4] [3]